IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

Merge Path – Cache-Efficient Parallel Merge and Sort

Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, Yitzhak Birk

CCIT Report #802
January 2012

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

CCIT Report #802 January 2012

Merge Path – Cache-Efficient Parallel Merge and Sort

Saher Odeh, Oded Green ‡, Zahi Mwassi, Oz Shmueli, Yitzhak Birk
Electrical Engineering Department
Technion
Haifa, Israel
{sahero, ogreen, zahim}@tx.technion.ac.il, {shmueli, birk}@ee.technion.ac.il

***Abstract*—Merging two sorted arrays is a prominent building block for sorting and other functions. Its efficient parallelization requires balancing the load among compute cores, minimizing the extra work brought about by parallelization, and minimizing inter-thread synchronization requirements. Due to the extremely low compute to memory-access ratio, it is also critically important to efficiently utilize the memory system: minimize memory traffic, maximize the cache hit rate and minimize cache-coherence related activity.**

We present a novel approach to partitioning the two sorted arrays into pairs of contiguous sequences of elements, one from each array, such that 1) each pair comprises any desired total number of elements, and 2) the elements of each pair form a contiguous sequence in the final merged sorted array. While the resulting partition and the computational complexity are similar to those of certain previous algorithms, our approach is different, extremely intuitive, and offers interesting insights. Based on this, we present a synchronization-free, cache-efficient merging (and sorting) algorithm. While we use CREW PRAM as the basis, our algorithm is easily adaptable to additional architectures. In fact, our approach is even relevant to sequential cache-efficient sorting. The new algorithm has been implemented both on the HyperCore many-core shared-cache architecture and on a sizable x86 system, with emphasis on cache efficiency. The algorithms and performance results are presented, along with important cache-related insights.

Keywords-component; Cache Memories; Parallelism and concurrency; Parallel processors; Sorting and searching

I. INTRODUCTION

Merging two sorted arrays, A and B, to form a sorted array S is an important utility, and is the core the of merge-sort algorithm [1]. The merging (e.g., in ascending order) is carried out by repeatedly comparing the smallest (lowest-index) as-yet unused elements of the two arrays, and appending the smaller of those to the result array.

Given an (unsorted) N-element array, merge-sort comprises a sequence of $\log_2 N$ rounds: in the first round, $N/2$ disjoint pairs of adjacent elements are sorted, forming $N/2$ sorted arrays of size two. In the next round, each of the $N/4$ disjoint pairs of two-element arrays is merged to form a sorted 4-element array. In each subsequent round, array pairs are similarly merged, eventually yielding a single sorted array.

Consider the parallelization of merge-sort using p compute cores (or processors or threads, terms that will be used synonymously). Whenever $|S| = N \gg p$, the early rounds are trivially parallelizable, with each core assigned a subset of the array pairs. This, however, is no longer the case in later rounds, as only few arrays remain. Because the total amount of computation is the same for all rounds, effective parallelization thus requires the ability to parallelize the merging of two sorted arrays.

An efficient Parallel Merge algorithm must have several salient features, some of which are required due to the very low compute to memory-access ratio: 1) equal amounts of work for all cores; 2) minimal inter-core communication (platform-dependent ramifications); 3) minimum excess work (for parallelizing, as well as replication of effort); and 4) efficient access to memory (high cache hit rate and minimal cache-coherence overhead). Coherence issues may arise due to concurrent access to the same address, but also due to concurrent access to different addresses in the same cache line (false sharing). Memory issues have platform-dependent manifestations.

The naïve approach to parallel merge entails partitioning each of the two arrays into equal-length contiguous sub-arrays and assigning a pair of same-numbered sub arrays to each core. Each core then merges its pair to form a single sorted array, and those are concatenated to yield the final result. Unfortunately, this is incorrect. (To see this, consider the case wherein all the elements of A are greater than all those of B.) So, correct partitioning is the key to success.

In this paper, we present a parallel merge algorithm for Parallel Random Access Machines (PRAM), namely shared-memory architectures that permit concurrent (parallel) access to memory. PRAM systems are further categorized as CRCW, CREW, ERCW or EREW, where C, E, R and W denote concurrent, exclusive, read and write, respectively. Our algorithm assumes CREW, but can be adapted. Also, complexity calculations assume equal access time of any core to any address, but this is not a requirement.

Our algorithm is load-balanced, lock-free, requires negligible excess work, and is extended to a memory-efficient version. Being lock-free, the algorithm does not rely on a set of atomic instructions of any particular platform. The efficiency of memory access is also not confined to one kind of architecture; in fact, the memory access is efficient for both private- and shared-cache architectures.

‡*Oded Green is currently with the School of Computational Science and Engineering at Georgia Tech, GA 30332. This work was done while Oded was at the Technion.*

We show a correspondence between the merge operation and the traversal of a path on a grid, from the upper left corner to the bottom right corner and going only rightward or downward. This greatly facilitates the comprehension of parallel merge algorithms. By using this path, dubbed *Merge Path*, one can divide the work equally among the cores.

Our actual basic algorithm is similar to that of [2]. However, using insights from the aforementioned geometric correspondence, we develop a new cache-efficient algorithm, and use it for a memory-efficient parallel sort algorithm.

The remainder of the paper is organized as follows. In Section II, we present the Merge Path, the Merge Matrix and the relationship between them. These are used in Section III to develop parallel merge and sort algorithms. Section IV introduces cache-related issues and presents a cache-efficient merge algorithm. Section V discusses related work, Section VI presents implementations and performance results, and Section VII offers concluding remarks.

II. MERGE PATH

A. *Construction and basic properties*

Consider two sorted arrays, A and B, with $|A|$ and $|B|$ elements, respectively. Without loss of generality, assume that they are sorted in ascending order. As depicted in Fig. 1 (ignore the contents of the matrix), create a column comprising A's elements and a Row comprising B's elements, and an $|A|x|B|$ matrix M, each of whose rows (columns) corresponds to an element of A (B). We refer to this matrix as the *Merge Matrix*. A formal definition and additional details pertaining to M will be provided later.

Next, let us merge the two arrays: in each step, pick the smallest (regardless of array) yet-unused array element. Alongside this process, we construct the *Merge Path*. Referring again to Fig. 1, start in the upper left corner of the grid, i.e., at the upper left corner of $M[1,1]$. If $A[1]>B[1]$, move one position to the right; else move one position downward. Continue similarly as follows: consider matrix position (i,j) whose upper left corner is the current end of the merge path: if $A[i]>B[j]$, move one position to the right; else move one position downward; having reached the right or bottom edge of the grid, proceed in the only possible direction. Repeat until reaching the bottom right corner.

The following four lemmas follow directly from the construction of the Merge Path:

Lemma 1: Traversing a Merge Path from beginning to end, picking in each rightward step the smallest yet-unused element of B, and in each downward step the smallest yet-unused element of A, yields the desired merger. □

Lemma 2: Any contiguous segment of a Merge Path is composed of a contiguous sequence of elements of A and of a contiguous sequence of elements of B. □

Lemma 3: Non-overlapping segments of a merge path are composed of disjoint sets of elements, and vice versa. □

Lemma 4: Given two non-overlapping segments of a merge path, all array elements composing the later segment are greater than or equal to all those in the earlier segment.

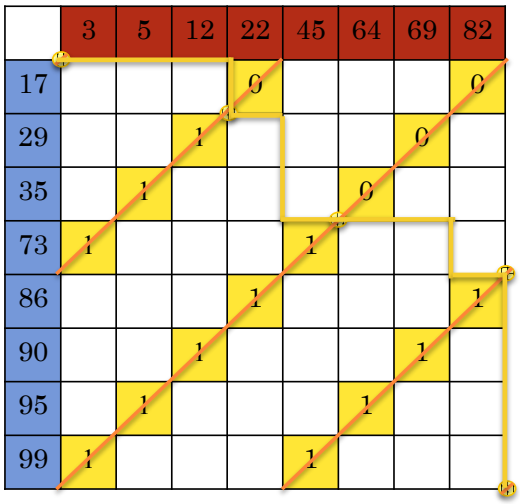

Figure 1 - The cross diogonals in a Merge Matrix are used to find the points of change between the ones and the zeros, i.e., the intersections with the Merge Path.

Theorem 5: Consider a set of element-wise disjoint sub-array pairs (one, possibly empty sub-array of A and one, possibly empty sub-array of B), such that each such pair comprises all elements that, once sorted, jointly form a contiguous segment of a merge path. It is claimed that these array pairs may be merged in parallel and the resulting merged sub-arrays may be concatenated according to their order in the merge path to form a single sorted array.

Proof: By Lemma 1, the merger of each sub-array pair forms a sorted sub-array comprising all the elements in the pair. From Lemma 2 it follows that each such sub-array is composed of elements that form a contiguous sub-array of their respective original arrays, and by Lemma 3 the given array pairs correspond to non-overlapping segments of a merge path. Finally, by Lemma 4 and the construction order, all elements of a higher-indexed array pair are greater than or equal to any element of a lower-indexed one, so concatenating the merger results yields a sorted array.

Corollary 6: Any partitioning of a given Merge Path of input arrays A and B into non-overlapping segments that jointly comprise the entire path, followed by the independent merger of each corresponding sub-array pair and the concatenation of the results in the order of the corresponding Merge-Path segment produces a single sorted array comprising all the elements of A and B. □

Corollary 7: Partitioning a Merge Path into equisized segments and merging the corresponding array pairs in parallel balances the load among the merging processors.

Proof: each step of a Merge Path requires the same operations (read, compare and write), regardless of the outcome. □

Equipped with the above insights, we next set out to find an efficient method for partitioning the Merge Path into equal segments. The challenge, of course, is to do so without actually constructing the Merge Path, as its construction is

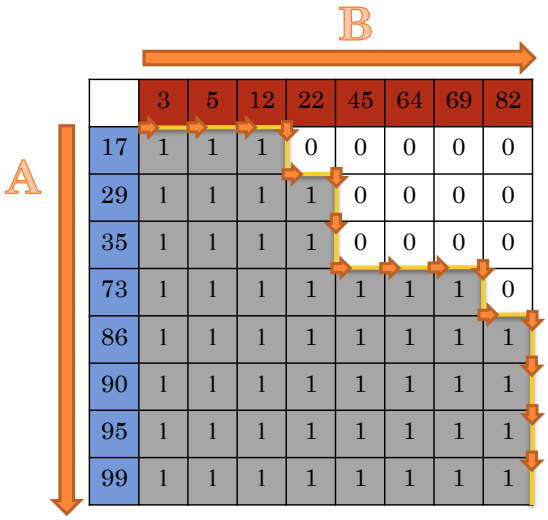

Figure 2 - Merge Matrix and Merge Path.

equivalent to carrying out the entire merger. Once again using the geometric insights provided by Fig. 1, we begin by exposing an interesting relationship between positions on any Merge Path and cross diagonals (ones slanted upward and to the right) of the Merge Matrix *M*. Next, we define the contents of a Merge Matrix and expose an interesting property involving those. With these two building blocks at hand, we construct a simple method for parallel partitioning of any given Merge Path into equisized segments. This, in turn, enables parallel merger.

B. The Merge Path and cross diagonals

Lemma 8: Regardless of the route followed by a Merge Path, and thus regardless of the contents of *A* and *B*, the *i*'th point along a Merge Path lies on the *i*'th cross diagonal of the grid and thus of the Merge Matrix *M*.

Proof: each step along the Merge Path is either to the right or downward. In either case, this results in moving to the next cross diagonal. □

Theorem 9: Partitioning a given merge path into p equisized contiguous segments is equivalent to finding its intersection points with *p-1* equispaced cross diagonals of *M*,

Proof: follows directly from Lemma 8. □

C. The Merge Matrix – content & properties

Definition 1: A binary merge matrix M of A, B is a Boolean two dimensional matrix of size $|A| \times |B|$ such that:

$$M[i,j] = \begin{cases} 1 & A[i] > B[j] \\ 0 & otherwise \end{cases}.$$

Proposition 10: Let M be a binary merge matrix. Then $[i,j] = 1 \Rightarrow \forall k,m: i \le k \le |A| \wedge 1 \le m \le j, M[k,m] = 1$

Proof: If $M[i,j] = 1$ then according to definition 1, $A[i] > B[j]$. $k \ge i \Rightarrow A[k] \ge A[i]$ (A is sorted). $j \ge m \Rightarrow B[j] \ge B[m]$ (B is sorted). $A[k] \ge A[i] > B[j] \ge B[m]$ and according to definition 1, $M[k,m] = 1$. □

Proposition 11: Let M be a binary merge matrix. If $M[i,j] = 0$, then $\forall k,m: 1 \le k < i \wedge j \le m \le |B|$, $M[k,m]$ is false.

Proof: Similar to the proof of proposition 10. □

Corollary 12. The entries along any cross diagonal of *M* form a monotonically non-increasing sequence. □

D. The Merge Path and the Merge Matrix

Having established interesting properties of both the Merge Path and the Merge Matrix, we now relate the two, and use *P(M)* to denote the Merge Path corresponding to Merge Matrix *M*.

Proposition 13: Let (i,j) be the highest point on a given cross diagonal M such that $M[i,j-1] = 1$ if exists, otherwise let (i,j) be the lowest point on that cross diagonal. Then, $P(M)$ passes through (i,j). This is depicted in Figure 2.

Proof: by induction on the points on the path.

Base: The path starts at $(1,1)$. The cross diagonal that passes through $(1,1)$ consists only of this point; therefore, it is also the lowest point on the cross diagonal.

Induction step: assume the correctness of the claim for all the points on the path up to the point (i,j). Consider the next point on $P(M)$. Since the only permissible moves are R, D, the next point can be either $(i,j+1)$ or $(i+1,j)$, respectively.

Case 1: R move. The next point is $(i,j+1)$. According to Definition 1, $M[i,j] = 1$. According to the induction assumption, either $i = 1$ or $M[i-1,j] = 0$. If $i = 1$ then the new point is the highest point on the new cross diagonal such that $M[i,j] = 1$. Otherwise, $M[i-1,j] = 0$. According to Proposition 11, $M[i-1,j+1] = 0$. Therefore, $(i,j+1)$ is the highest point on its cross diagonal at which $M[i,j] = 1$.

Case 2: the move was D, then the next point is $(i+1,j)$. According to Definition 1, $M[i,j] = 0$. According to the induction assumption, either $j = 1$ or $M[i,j-1] = 1$. If $j = 1$ then the new point is the lowest point in the new cross diagonal. Since $M[i,j] = 0$ and according to Proposition 11, the entire cross diagonal is 0. Otherwise, $M[i,j-1] = 1$. According to Proposition 10, $M[i+1,j-1] = 1$. Therefore, $(i,j+1)$ is the highest point on its cross diagonal at which $M[i+1,j-1] = 1$. □

Theorem 14: Given sorted input arrays A and B, they can be partitioned into p pairs of sub-arrays corresponding to p equisized segments of the corresponding merge path. The *p-1* required partition points can be computed independently of one another (optionally in parallel), in at most $\log_2(\min(|A|,|B|))$ steps per partition point, with neither the matrix nor the path having actually been constructed.

Proof: According to Theorem 9, the required partition points are the intersection points of the Merge Path with *p-1* equispaced (and thus content-independent) cross diagonals

of M. According to Corollary 12 and Proposition 13, each such intersection point is the (only) transition point between '1's and '0's along the corresponding cross diagonal. (If the cross diagonal has only '0's or only '1's, this is the uppermost and the lower most point on it, respectively.) Finding this partition point can be done by way of a binary search, whereby in each step a single element of A is compared with a single element of B. Since the length of a cross diagonal is at most $\min(|A|,|B|)$, at most $\log_2(\min(|A|,|B|))$ steps are required. Finally it is obvious from the above description that neither the Merge Path nor the Merge Matrix needs to be constructed and that the *p-1* intersection point can be computed independently and thus in parallel. □

III. Parallel Merge and Sort

Given two input arrays A and B parallel merger is carried out by p processors as follows:

Algorithm 1 – Parallel Merge (*A,B,p*)

/* i = processor number in the range 1..p */

In parallel do:

1. DiagonalNum=$(i-1)\cdot(|A|+|B|)/p$+1
2. Compute intersection of the merge path with the relevant diagonal //Binary search
3. Execute $(|A|+|B|)/p$ steps of sequential merge, writing the results to output array positions starting at $(i-1)\cdot(|A|+|B|)/p$+1

Barrier.

Remark. Note that no communication is required among the cores: they write to disjoint sets of addresses and, with the exception of reading in the process of finding the intersections between the Merge Path and the diagonals, read from disjoint addresses. Whenever $|A|+|B| >> p$, which is the common case, this means that concurrent reads from the same address are rare.

Summarizing the above, the time complexity of the algorithm for $|A|+|B|=N$ and p processors is given by $O(N/p + \log(N))$, and the work complexity is given by $O(N + p \cdot \log N)$. For $p < N/log(N)$, this algorithm is considered to be optimal. In the next section, we will further address the issue of efficient memory (cache) utilization.

Finally, merge-sort can be used, employing Parallel Merge to carry out each of $\log_2 N$ rounds. The rounds are carried out one after the other.

The time complexity of this Parallel Merge-Sort is:

$$O(N/p \cdot log(N/p) + N/p \cdot log(p) + \log(p) \cdot \log(N)) = O(N/p \cdot \log(N) + \log(p) \cdot \log(N))$$

In the first expression, the first component corresponds to the sequential sort carried out concurrently by each core on *1/p* of the input., and the two remaining ones correspond to the subsequent rounds of parallel merges.

IV. Cache Efficient Merge Path

A. Overview

The rate at which merging and sorting can be performed even in memory (as opposed to disk), is often dictated by the performance of the memory system rather than by processing power. This is due to the fact that these operations require a very small amount of computing per unit of data, and the fact that only a small amount of memory, the cache, is reasonably fast. (The next level in the memory hierarchy typically features a ten-fold higher access latency as well as coarser memory-management granularity.) Parallel implementation on a shared memory system further aggravates the situation for two reasons: 1) the increased compute power is seldom matched by a commensurate increase in memory bandwidth, at least beyond the 1st-level or 2nd-level cache, and 2) cache coherence mechanisms can present an extremely high overhead. In this section, we address the memory issues.

Assuming large arrays (relative to cache size) and merge-sort, it is clear that data will have to be brought in again for each of the $\log_2 N$ rounds of the sorting algorithm, so we again focus on merging a pair of sorted arrays.

In the remainder of this section, we examine the cache efficiency issue in conjunction with our algorithm, offering important insights, exploring trade-offs and presenting our approaches. Before continuing along this path, however, let us digress briefly to discuss relevant salient properties of hierarchical memory in general, and particularly in shared-memory environments.

B. Memory-hierarchy highlights

Cache Organization and management

Unlike software-managed buffers, caches do not offer the programmer direct control over their content and, specifically, over the choice of item for eviction. Furthermore, in order to simplify their operation and management, caches often restrict the locations in which an item may reside based on certain bits of its original address (the index bits). The number of cache location at which an item with a given address may reside is referred to as the level of associativity: in a fully associative cache there are no restrictions; at the other extreme, a direct-mapped cache permits any given address to be mapped only to a single specific location in the cache. The collection of cache locations to which a given address may be mapped is called a *set*, and the size of the set equals the *degree of associativity*.

Whenever an item must be evicted from the cache in order to make room for a new one, the cache management system must select an item from among the members of the relevant set. One prominent replacement policy is *least recently used* (LRU), whereby the evicted item is the set member that was last accessed in the most distant pass. Another is *first in – first out* (FIFO), whereby the evicted

item is the one that was last brought into the cache in the most distant past. Additional considerations may include eviction of pages that have not been modified while in the cache, as they often don't have to be copied to the lower level in the hierarchy, as it maintains a copy (an *inclusive* cache hierarchy).

Cache content is managed in units of cache line. We will initially assume that the size of an array item is exactly one cache line, but will later relax this assumption.

Cache performance

The main cache-performance measure is the *hit rate*, namely the fraction of accesses that find the desired data in the cache. (Similarly, *miss rate = 1- hit rate.)*

There are three types of cache misses: Compulsory, Capacity, and Contention [3].

1) Compulsory – a miss that occurs upon the first request for a given data item. (Whenever multiple items fit in a cache line, as well as when automatic prefetching is used, the compulsory miss rate may be lower than expected. (Access to contiguous data would result in one miss per cache line or none at all, respectively.)

2) Capacity – this refers to cache misses that would have been prevented with a larger cache.

3) Conflicts – these misses occur despite sufficient cache capacity, due to limited flexibility in data placement (limited associativity and non-uniform use of different sets).

Cache coherence

In multi-core shared-memory systems with private caches, yet another complication arises from the fact that the same data may reside in multiple private caches (for reading purposes), yet coherence must be ensured when writing. There are hardware cache-coherence mechanisms that obviate the programmer's need to be concerned with correctness; however, the frequent invocation of these mechanisms can easily become the performance bottleneck. The most expensive coherence-related operations occur when multiple processors attempt to write to the same place. The fact that management and coherence mechanisms operate at cache-line granularity complicates matters, as coherence-related operations may take place even when cores access different addresses, simply because they are in the same cache line. This is known as false sharing.

We next present two approaches for cache-efficient parallel merge, and discuss them in the context of the various aforementioned realities of cache systems.

C. *Cache-Efficient Parallel Merge*

Collisions in the cache are avoided when different items are guaranteed to be able to reside in different cache locations, as well as when they are in the cache (and are actually still needed) at different times. In a Merge operation, a cache-resident item is usually required for a very short time, and is used only once. However, many items are brought into the cache. Also, the relative addresses of "active" items are data dependent. This is true among

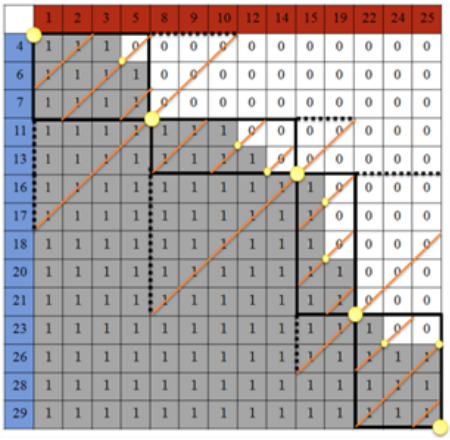

Figure 3 - Merge Matrix for the cache efficient algorithm. The yellow circles depict the initial and final points of the path for a specific block in the cache algorithm.

elements of different arrays (*A, B, S)* and, surprisingly, also among same-array elements accessed by different cores. This is because the segment-partition points in any given array are data dependent, as is the rate at which its elements are consumed.

Given our efficient parallelization, we are able to efficiently carry out parallel merger of even cache-size arrays. In view of this, we explore approaches that ensure that all elements that may be active at any given time can co-reside in cache.

Let *C* denote cache size (in elements). Our general approach is to break the overall merge path into cache-size (actually a fraction of that) segments, merging those segments one after the other, with the merging within each segment being parallelized. We refer to this as *Segmented Parallel Merge*, SPM. See Fig. 3.

The starting point of the segment handled in each iteration, i.e., the indices of the first elements of *A* and *B* to be considered, is known. Unlike in a full parallel merger, however, the last elements to be considered are not necessarily known. (Each core does know how many elements of *S* it should produce, so the question is only how many elements to fetch from each array.) We next present two schemes, common to which is the fact that at all times the amount of memory needed for the parallel merge does not exceed the cache size, and if the level of associativity is at least 3, collision freedom can be guaranteed.

Scheme I: Fetch and Delimit (FaD)

Lemma 15. A merge-path segment of length *L* comprises at most *L* consecutive elements of *A* and at most *L* consecutive elements of *B*. □

Theorem 16. Given *L* consecutive elements of *A* and *L* consecutive elements of *B,* starting with the first element of each of them in the segment being constructed, one can compute in parallel the *p* segment starting points so as to enable *p* consecutive segments of length *L/p* to be constructed in parallel.

Proof: Consider the *p-1* cross diagonals of the merge matrix comprising the aforementioned elements of the two arrays, such that the first one is *L/p* away from the upper left corner and the others are spaced with the same stride. The farthest cross diagonal will require the *L*'th provided element from each of the two arrays, and no other point along any of the diagonals will require "later" elements. Also, since the farthest diagonal is at distance *L* from the upper left corner (Manhattan distance), the constructed segment will be of length *L*. □

Remark. Unlike the case of a full merge of two sorted arrays of size *L*, not all elements will be used. While *L* elements will be consumed in the construction of the segment, the mix of elements from *A* and from *B* is data dependent.

In order to avoid the extra complexity of using the same space for input elements and for merged data, let *L=C/3*, where *C* is the cache size.

Algorithm 2 – Fetch and Measure (FaD)

Repeat the following (*|A|+|B|)/L* times /* *L=|C|/3* */

1. If first iteration, fetch the first *L* items of *A* and *B;* Else fetch the next elements of *A* and *B* in numbers equal to the number respective consumed elements in the previous iteration, overwriting the used elements of the respective arrays (cyclic buffer).
2. Parallel do:
    a. Find the core's segment starting point /* binary search on cross diagonal */
    b. Merge (sequential) *L/p* steps, commencing at the start point.
3. Write the results out to memory.

Implementation issues

By construction, the cache size suffices for the parallel construction of each segment. Also, data is brought in only once, and sequentially, so compulsory misses are held to the (platform dependent) bare minimum.

Limited associativity

Proposition 17. with 3-way associativity or higher, conflict misses can be avoided.

Proof: With *k*-level associativity, any consecutive *C/k* addresses are mapped to *C/k* different sets. The *C/3* items of each of *A,B* and the merged array will take up exactly one position in each of the three sets, regardless of the start address of each of these element sequences. Similarly, each will take up two positions in a 6-way set associative cache, three in a 9-way, etc. For associativity levels that are greater than 3 but not integer multiples thereof, one can reduce the segment length such that each array's elements occupy at most a safe number of positions in each set. (A safe number is one such that even if all three arrays occupy the maximum number of positions in a given set, this will not exceed the set size, i.e., the degree of associativity.) □

Cache line containing multiple elements

This may result in more data than planned being brought in. The simple remedy is to slightly reduce the segment size, thereby guaranteeing that the total amount will not exceed the allocated space.

There may also be cache coherence issues; these will be addressed together for the two schemes.

Cache replacement policy

A problem may arise at replenishment time. Consider, for example, LRU and a situation wherein a given merge segment only comprises elements of *A*. As replenishment elements are brought in to replace the used elements of *A,* the least recently used elements are actually those of *B*, as both the *A* element positions and the result element positions were accessed in the previous iteration whereas only one element of *B* was accessed (it repeatedly "lost" in the comparison). A similar problem occurs with a FIFO policy.

A possible solution for LRU is, prior to fetching replenishment elements, touching all cache lines containing unused input elements. If each cache line only contains a single item, this would represent approximately a 50% overhead in cache access (the usual comparison is between the loser of the previous comparison, which is in a register, and the next element of the winning array, which must be read from cache; also, the result must be written. So the number of cache accesses per step grows from 2 to 3.) If there are multiple elements per cache line, the overhead quickly becomes negligible.

Scheme II: Delimit and Fetch (DaF)

Here, given the starting point of the current iteration of the (sequentially executed) outer loop, we (sequentially) compute the intersection point of the overall merge path with the diagonal that will terminate the segment being constructed; i.e., with the cross diagonal that is at Manhattan distance *L* from that in which the starting point resides. Next, we fetch exactly the required elements for the (parallel) construction of the next segment and carry out the merger as as in FaD. In fact, outer-loop segment boundaries may be computed in parallel as in the cache-agnostic parallel merge algorithm, and their values stored until needed.

Algorithm 3 – Delimit and Fetch (DaF)

Repeat the following (*|A|+|B|)/L* times /* *L=C/3* */

1. If first iteration, compute the end point of the first segment of length *L;* Else, determine the end point of the next length-*L* segment, using only as-yet unused array elements in finding the intersection point /*binary search on the appropriate cross diagonal of *M* */
2. Fetch the next elements of *A* and *B* as required for the construction of the next merge segment.
3. Parallel do:
    a. Find the core's segment starting point /* binary search on a cross diagonal */

b. Merge (sequential) *L/p* elements, commencing at the start point.

4. Write the results out to memory.

Implementation issues are mostly similar to those of DaF and are omitted for brevity. Exceptions will be discussed next as we compare the schemes.

Scheme comparison

One apparent advantage of DaF over FaD is that, since we only fetch the elements that will be needed, the segment length in each iteration can be *C/2* rather than *C/3;* sub-segment length is 1.5x larger, thereby commensurately reducing the overall required number of segment-boundary computations. (Each boundary computation may require one additional iteration but this is negligible.)

With **limited associativity**, however, this is no longer the case, as demonstrated next with a 3-way set associative cache.

Consider a segment of size *C/2* comprising *(C/2-1)* consecutive elements of *A* and a single element of *B*. The output segment also comprises *C/2* consecutive elements. Here, the elements of *A* occupy two slots in some of the sets, the element of *B* may be mapped to one of those, and so may up to two elements of the output array, causing conflict misses. With 2-way set associativity, elements of *A* will occupy at most one slot in any set, as will elements of the output segment. However, the element(s) of *B* may still be mapped to sets that already have to elements, again causing a collision. Consequently, here too the segment size may be at most *C/3*. Further elaborations for different levels of associativity are omitted for brevity.

Implications of the **Cache replacement policy** expose the main, perhaps only, advantage of DaF over FaD. In each iteration, all fetched elements are consumed and are no longer needed, so there is no need to touch cache lines in order to prevent their premature eviction. FaD thus works well with both LRU and FIFO.

Remark. In both FaD and DaF, one can either write to the output array in each step or first construct a bitmap denoting from which array to take the next element, and copy onto the output array once done. The former approach appears more efficient. The exception is whenever an item contains a sorting key as well as additional data, and these need to be stored contiguously in the output array.

We conclude this section by deriving the computational complexity (total amount of work) and the time complexity of the cache-efficient merge. We will do it for DaF, but the complexity of FaD is similar.

Computational complexity (DaF)

Assuming a total segment size of *L=C/3* per sequential iteration of the algorithm, there are *3N/C* such iterations, In each of those, only *2L=2C/3* need to be considered in order to determine the end of the segment and, accordingly, the elements that should be copied into the cache. Because the sub-segments of this segment are to be created in parallel, each of the p cores must compute its starting points (in A and in B) independently. If this is done following the determination of the aforementioned ending point, it need only consider L elements; otherwise it must consider $2L$. For simplicity, we assume the latter. Finally, the complexity of the merge itself is N.

The computational complexity of the cache-efficient merge of N elements given a cache of size C and p cores is:

$$O(N/C \cdot p \cdot logC + N).$$

Normally, $p \ll C \ll N$, in which case this becomes $O(N)$. In other words, the parallelization overhead is negligible.

The time complexity is

$$O\big(N/C \cdot (logC + C/p)\big).$$

Neglecting $logC$ relative to C/p, this becomes $O(N/p)$, which is optimal. Finally, looking at typical numbers and at the actual algorithms, it is evident that the various constant coefficients are very small, so this is truly an extremely efficient parallel algorithm and the overhead of partitioning into smaller segments is insignificant.

D. Cache-Efficient Parallel Sort

Initially, partition the unsorted input array into equisized sub-arrays whose size is some fraction of the cache size C.

Next, iterate over these sub-arrays, sorting them one by one using the parallel sort algorithm on all p processors as explained in an earlier section.

Finally, proceed with merge rounds; in each of those, the cache-efficient parallel merge algorithm is applied to every pair of sorted sub-arrays. This is repeated until a single array is produced.

We now derive the time complexity of the cache efficient parallel sort algorithm. We divide the complexity into two stages: 1) the complexity of the parallel sorting of the sub-arrays of at most C elements, and 2) the complexity of the cache-efficient merge stages.

In the first stage, depicted in Fig. 4, the parallel sort algorithm is invoked on the cache sized sub-arrays. The number of those sub-arrays is $O(N/C)$. Hence, the time complexity of this stage is $O\big(N/C \cdot (C/p \cdot \log(C) + \log(p) \cdot \log(C))\big)$.

The second stage may be viewed as a binary tree of merge operations. The tree leaves are the sorted cache sized sub-arrays. Each two merged sub-arrays are connected to the merged sub-array, and so on. The complexity of each level in the tree is $O(N/p + N/C \cdot \log(p))$. The height of the tree is $O(\log N/C)$. Hence, this stage's complexity is $O\big(\log(N/C) \cdot (N/p + N/C \cdot \log(p))\big)$.

The total complexity of the cache-efficient parallel sort algorithm is the summation of the complexities of the two

Figure 4 - Cache-efficient parallel sort first stage. Each cache sized block is sorted followed by parallel merging

stages, which yield: $O(N/p \cdot \log(N) + N/C \cdot \log(p) \cdot \log(C))$.

One may observe again that the new algorithm has a slightly higher complexity, $N/C \cdot \log(C) \cdot \log(p) > \log N \cdot \log(p)$, due to the numerous partitioning stages, however for system that a cache miss is expensive, this increase in complexity may be justified.

V. Related Work

In this section, we review previous works on the subjects of parallel sorting and parallel merging, and relate our work to them.

Prior works fall into two categories: 1) algorithms that use a problem-size dependent number of processors, and 2) algorithms that use a fixed number of processors.

Several algorithms have been suggested for parallel sorting. While parallel merge can be a building block for parallel sorting, some of the parallel sorting algorithms do not require merging. An example is Bitonic Sort [4] in which $O(N \cdot (\log N)^2)$ comparators are used ($N/2$ comparators are used in each stage) to sort N elements in $O((\log N)^2)$ cycles. Bitonic sort falls into the aforementioned first category. Our work is in the latter.

We consider two complexity measures: 1) time complexity (the time required to complete the task), and 2) overall work complexity, i.e, the total number of basic operations carried out. In a load balanced algorithm like ours, the work complexity is the product of time complexity and the number of cores. Even with perfect load balancing, however, one must be careful not to increase the total amount of work (overhead, redundancy, etc.), as this would increase the latency. Similarly, one must be careful not to introduce stalls (e.g., for inter-processor synchronization), as these would also increase the elapsed time even if the "net" work complexity is not increased.

Merging two sorted arrays requires $\Omega(N)$ operations. Some of the parallel merging algorithms, including ours, have a work complexity of $O(N + p \cdot \log N)$. For $p \leq N/\log N$, the latter component is negligible and the complexity is $O(N)$, as observed in [5]. Also, there are no synchronization stalls in our algorithm.

In [6], as in our work, a $CREW\ PRAM$ memory model is used. There, a mechanism for partitioning the workload is presented. This mechanism is less efficient than ours and does not feature perfect load balancing; although each processor is responsible for merging $O(N/p)$ elements on average, a processor may be assigned as many as $2N/p$ elements. This can introduce a stall to some of the cores since all the cores have to wait for the heaviest job. For truly efficient algorithms, namely ones in which the constants are also tight, as is the case with our algorithm, such a load imbalance can cause a 2X increase in latency! The time complexity of this algorithm is $O(1 + \log p + \log N + N/p)$. For $N \gg p$, which is the case of interest, it is $O(N/p + \log N)$.

In [5], Akl and Santoro present a merging algorithm that is memory-conflict free using the $EREW$ model. It begins by finding one element in each of the given sorted arrays such that one of those two elements is the median (mid-point) in the output array. The elements found $(A[i], B[j])$ are such that if $A[i]$ is the aforementioned median then $B[j]$ is the largest element of B that is smaller than $A[i]$ or the smallest element of B that is greater than $A[i]$. Once this median point has been found, it is possible to repeat this on both sets of the sub-arrays. Their way of finding the median is similar to the process that we use. The complexity of finding the median is $O(\log(N))$. As these arrays are non-overlapping, there will not be any more conflict on accessed data. This stage is repeated until there are p partitions. This requires $O(\log(p))$ iterations. Once all the partitions have been found, it is possible to merge each pair of sub-arrays sequentially, concurrently for all pairs, and to simply concatenate the results to form the merged array. The overall complexity of this algorithm is $O(N/p + \log N \log p)$. The somewhat higher complexity is the price for the total elimination of memory conflicts.

In [2], an algorithm that is conceptually similar to that of [5] is presented. They initially present an algorithm that finds one element in each of two given sorted arrays such that one of these elements is $k-th$ smallest element in the output (merged) array. In [5] they start off by finding $k = N/2$. In [2], the elements sought after are those that are equispaced (N/p positions apart) in the output array. Finding each of these elements has the complexity of $O(\log(N))$. This algorithm is aimed for $CREW$ systems. The complexity of this algorithm is $O(N/p + \log N)$.

Our algorithm is very similar to the one presented in [2]. However, our approach is different in that we show a correspondence between finding the desired elements and finding special points on a grid. Finally, using this correspondence along with additional insights and ideas, we also provide cache efficient algorithms for parallel merging and sorting that did not appear in any of the related works.

The work done in [7] is an extension of [2], in which the algorithm is adapted to an $EREW$ machine with a slightly increased complexity of $O(N/P + \log N + \log P)$.

Merging and sorting using GPUs is a topic of great interest as well, and raises additional challenges that need to be addressed. In [8] a radix sort for the GPU is presented. In addition to the radix sort, the authors suggest a merge-sort algorithm for the GPU, in which the a pair-wise merge tree is required in the final stages. In [9], a hybrid sorting algorithm is presented for the GPU. Initially the data is sorted using bucket sort and this is followed by a merge sort. The bucket approach suffers from workload imbalance and requires atomic instructions (i.e., synchronization).

Another focus of sorting algorithms is finding a way to implement them in a cache oblivious [10] way. As the algorithm in this paper focused on a) the merging stage and

not the entire sort, b) presented a cache aware merging algorithm, we will not elaborate on cache oblivious algorithms. The interested reader is referred to [11-13].

VI. Implementation and Measurement Results

According to Amdahl's Law [14][9], a fraction of an algorithm that is not parallelized limits the possible speedup, It is quite evident from the previous sections that we have succeeded in truly parallelizing the entire merging and sorting process, with negligible overhead for any numbers of interest. Nonetheless, we wanted to obtain actual performance results on real systems, mostly in order to find out whether there are additional issues that limit performance. Also, so doing increases the confidence in the theoretical claims.

We implemented our basic Parallel Merge algorithm an "on demand" variant of the DaF cache-efficient version. In the latter, the two arrays are segmented as described such that each segment-pair fits into a 3-way associative cache with no collisions, and the merging of one such segment pair begins only the merging of the previous pair has been completed. However, the actual fetching of array elements into the cache is done only once demanded by the processor, though any prefetch mechanisms of the system may kick in.

The algorithms were implemented on two very different platforms: HyperCore, a novel shared-cache many-core architecture by Plurality, and an dual 6-core processor Intel x86 system. We begin with a brief overview of the two systems systems, including system specifications, and then present some of the practical challenges of implementing the algorithms on each of the platforms. Following this, we present the speedup of both the new algorithms, regular Merge Path and the cache efficient version, on each of the systems. The runtime of Merge-Path with a single thread is used as the baseline.

A. *HyperCore Architecture*

Plurality's HyperCore architecture [15] features tens to hundreds of compute cores, interconnected to an even larger number of memory banks that jointly comprise the shared cache. The connection is via a high speed, low latency combinational interconnect. As there are no private caches for the cores, memory coherence is not an issue for CREW like algorithms. Same-address writes are serialized by the communication interconnect; however, for our algorithm this was not needed. The memory banks are equidistant from all the cores, so this is a UMA system. The shared cache has a number of memory banks that is larger than the number of cores in the system, reducing the number of conflicts on a single bank. Moreover, addresses are interleaved, so there are no persistent hot spots due to regular access patterns. The benefit of such an architecture is that there is no processor-cache communication bottleneck. Finally, the absence of private caches (and a large amount of state in them) and the UMA architecture permit any core to execute any compute task with equal

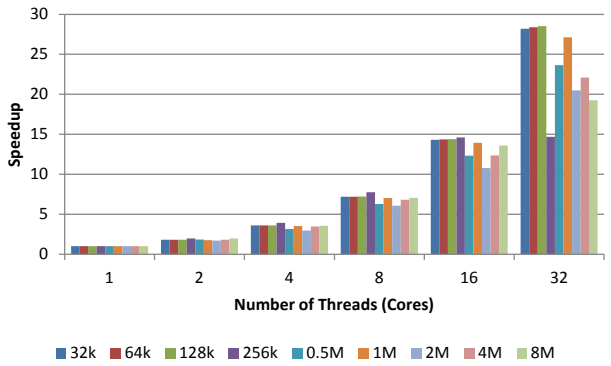

Figure 5 – Speedup of the regular merge path algorithm on Plurality.

efficiency. The memory hierarchy also includes off-chip (shared) memory. Finally, the programming model is a set of sequential "tasks" along with a set of precedence relations among them, and these are enforced by a very high throughput, low latency synchronizer/scheduler that dispatches work to the cores.

At the time of submission, Plurality has not manufactured the actual chip. We had access to an advanced experimental version of the HyperCore on an FPGA card. The FPGA version we used has a 1MB direct mapped cache and 32 cores. Furthermore, there was a latency issue on memory write back. Therefore, results are shown for an algorithm that does not write to memory. Instead, we saved the value in a private register.

We ran both the non-segmented and segmented versions of Parallel Merge Path with varying numbers of threads (cores). The input arrays (of type integer) tested on Plurality are substantially smaller than those that we tested on the x86-system due to the FPGA limitations. One might expect that merging smaller arrays would not offer significant speedups due to the overhead required in dispatching threads and to the fact that the search for partition points (binary search on a cross diagonal) become a more significant part of the computation. However, due to

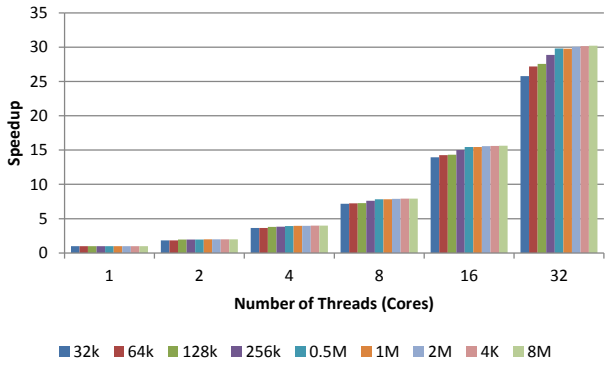

Figure 6 - Speedup of the cache-efficient merge path algorithm on Plurality; Segment size: *L=0.5MB=128K elements.*

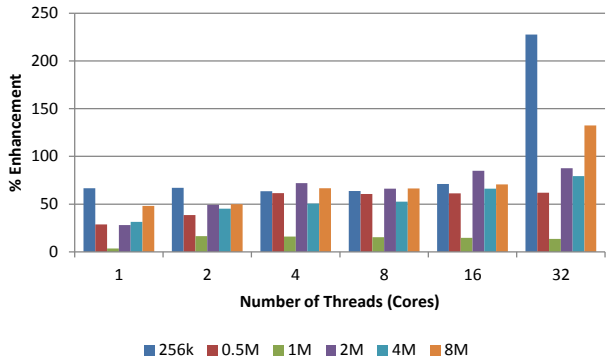

Figure 7 - The performance enhancement of the cache efficient algorithm over the regular algorithm on Plurality (%); Segment size: *L=0.5MB=128K elements.*

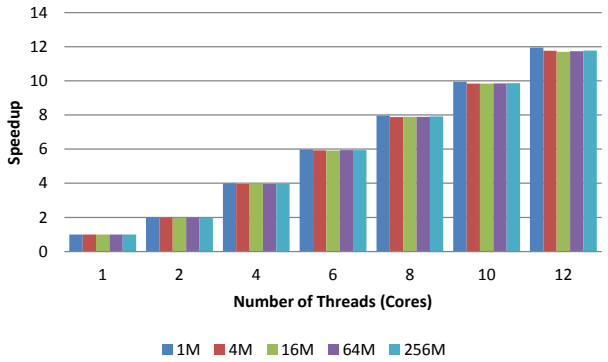

Figure 8 - Speedup of the regular Merge Path algorithm. Each of the colored bars represents a different sized input array. The sizes of the arrays are in Mega elements.

HyperCore's ability to dispatch a thread within a handful of cycles, the overhead is not a problem and makes the HyperCore an idle target platform. The sizes of the input arrays are denoted by the number of elements in each of the arrays A and B. This means that the total number of elements is $4 \cdot |A|$ and the size is $4 \cdot |A| \cdot |Integer|$. ($A$ and B are equisized, and $|S|=|A|+|B|$.)

Fig. 5 presents the speedup of our basic Parallel Merge algorithm with several numbers of cores. For each number of cores, results are shown for various input-array sizes. It is evident that they speedup is quite close to linear, and the array sizes do not matter much.

Fig. 6 similarly depicts the speedup for the segmented algorithm. Note, however that the cache is direct mapped, so collision freedom cannot be guaranteed. Nonetheless, the partition into sequential iteration, each carrying out parallel merge on a segment, does improve performance. The percentage speedup of the segmented version relative to same-parameter execution without segmentation is depicted in Fig. 7. The average improvement is by 80%.

Remark. Note that this is not directly related so speedup over a single core, as even single-core performance is affected. (In other words, Fig. 7 is independent of Fig. 5, 6.)

B. x86 System

We used a 2-processor, 2X6 core Intel X86 system with hyperthreading, It has L1 and L2 private caches for each core. The cores share an L3 cache. Because the cores have private caches, a cache coherency mechanism is required to ensure correctness. Furthermore, as we had multiple processors, each with its own L3 cache, the cache coherence mechanism had to communicate across processors; this is even more expensive from a latency point of view.

Specifically, we used a Dell-T610 server. The server consists of two X5670 INTEL processors, each of which having six cores with a private 32KB L1 data cache and a private 256KB L2 cache. Each processor has a 12MB L3 cache. The processors are connected via 6.4GT/s QPI. The server has 12GB DDR3 memory. For testing the algorithm, the following capabilities have been disabled: 1) INTEL hyper threading technology. 2) INTEL turbo technology. The reasons are fairly obvious.

Our implementation of Merge Path uses OpenMP. We tested the two algorithms using multiple sizes of integer arrays and different numbers of threads. In Fig. 8, the data set sizes refer to the size of each of the input arrays A and B. The output array S is twice this size, meaning that the total memory required for the 3 arrays is $4 \cdot |A| \cdot |type|$, where $|type|$ denotes the number of bytes need to stored the data type (for 32 bit integers this will be 4).

We show results of the regular (non segmented) merge path algorithm in Figure . Using the cache-efficient blocking algorithm on this platform yielded no improvement, apparently because the associativity of the caches is greater than what is required by all the threads in the system. On the contrary, the small overhead that is added by the cache-efficient algorithm yields a longer runtime.

In Figure we present the speedup of executing Merge Path using various size input arrays. One mega element refers to 2^{20} elements. As can be seen, the speedups are near linear, with a slight reduction in performance for the bigger input arrays: approximately $11.7X$ for 12 threads.

Remark. We note that the single-thread execution time of our algorithm was some 6% longer than a truly sequential merge algorithm. This is due in part to a few extra instructions, and possibly also to overhead of OpenMP.

VII. CONCLUSIONS

In this paper, we explored the issue of parallel sorting through the cornerstone of many sorting algorithms – the merging of two sorted arrays.

One important contribution of this paper is a very intuitive, simple and efficient approach to correctly partitioning each of two input sorted arrays into segments that, once pairs of segments, one from each, are merged, the

concatenation of the merged pairs yields a single sorted array. This partitioning is also done in parallel.

Another important contribution is an insightful consideration of cache related issues. This are extremely important because, especially when parallelized, sorting and merging are carried out at a speed that is very often determined by the memory subsystem rather than by the compute power. This culminated in cache-efficient parallel merging and sorting algorithms that also remain extremely efficient computationally.

Finally, we have actually implemented the algorithms on two very different platform: a multi-processor, multi-core X86 platform that represents mainstream computers, and Plurality's HyperCore many-core shared cache architecture, which perhaps comes as close as possible to a true CREW PRAM architecture. It is a brand new exciting architecture.

The performance measurements confirm the efficient parallelization, yet suggest that some issues are worthy of further study. Specifically, as traditional bottlenecks are broadened, new system components restrict performance. We believe that some may be related to memory management, others may involve the communication subsystem, and it may be that complex optimizations, often unpublished, may influence performance of a carefully designed program in unexpected ways.

Acknowlegements. We would like to thank Rob McColl of the HPC lab at Georgia Institute of Technology for his suggestions and lengthy discussions on MergePath; Peleg Aviely for his support pertaining to the Plurality HyperCore platform, and Shachar Raindel for his useful comments

REFERENCES

[1] T. H. Cormen*, et al.*, *Introduction to algorithms*. Cambridge, Mass.New York: MIT Press ;McGraw-Hill, 1990.

[2] N. Deo and D. Sarkar, "Parallel algorithms for merging and sorting," *Information Sciences,* vol. 56, pp. 151-161, 1991.

[3] J. L. Hennessy*, et al.*, *Computer architecture : a quantitative approach*, 4th ed. Amsterdam ; Boston: Morgan Kaufmann, 2007.

[4] K. E. Batcher, "Sorting Networks and Their Applications," presented at the AFIPS Conference Proceedings 1968.

[5] S. G. Akl and N. Santoro, "Optimal Parallel Merging and Sorting Without Memory Conflicts," *Computers, IEEE Transactions on,* vol. C-36, pp. 1367-1369, 1987.

[6] Y. Shiloach and U. Vishkin, "Finding the maximum, merging, and sorting in a parallel computation model," *Journal of Algorithms,* vol. 2, pp. 88-102, 1981.

[7] N. Deo*, et al.*, "An optimal parallel algorithm for merging using multiselection," *Information Processing Letters,* vol. 50, pp. 81-87, 1994.

[8] N. Satish*, et al.*, "Designing efficient sorting algorithms for manycore GPUs," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009, pp. 1-10.

[9] E. Sintorn and U. Assarsson, "Fast parallel GPU-sorting using a hybrid algorithm," *Journal of Parallel and Distributed Computing,* vol. 68, pp. 1381-1388, 2008.

[10] A. Aggarwal and S. V. Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM,* vol. 31, pp. 1116-1127, 1988.

[11] R. A. Chowdhury*, et al.*, "Oblivious algorithms for multicores and network of processors," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010, pp. 1-12.

[12] R. Cole and V. Ramachandran, "Resource Oblivious Sorting on Multicores Automata, Languages and Programming." vol. 6198, S. Abramsky*, et al.*, Eds., ed: Springer Berlin / Heidelberg, 2010, pp. 226-237.

[13] M. Frigo*, et al.*, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*, 1999, pp. 285-297.

[14] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," presented at the Proceedings of the April 18-20, 1967, spring joint computer conference, Atlantic City, New Jersey, 1967.

[15] "HyperCore Software Developer's Handbook," ed: Plurality, 2009.